MonkeyDB: Effectively Testing Correctness under Weak Isolation Levels

RANADEEP BISWAS, Informal Systems, France
DIPTANSHU KAKWANI, Microsoft, India
JYOTHI VEDURADA, IIT Hyderabad, India
CONSTANTIN ENEA, IRIF, University of Paris & CNRS, France
AKASH LAL, Microsoft Research, India

Modern applications, such as social networking systems and e-commerce platforms are centered around using large-scale storage systems for storing and retrieving data. In the presence of concurrent accesses, these storage systems trade off isolation for performance. The weaker the isolation level, the more behaviors a storage system is allowed to exhibit and it is up to the developer to ensure that their application can tolerate those behaviors. However, these weak behaviors only occur rarely in practice and outside the control of the application, making it difficult for developers to test the robustness of their code against weak isolation levels.

This paper presents MonkeyDB, a mock storage system for testing storage-backed applications. MonkeyDB supports a key-value interface as well as SQL queries under multiple isolation levels. It uses a logical specification of the isolation level to compute, on a read operation, the set of all possible return values. MonkeyDB then returns a value randomly from this set. We show that MonkeyDB provides good coverage of weak behaviors, which is complete in the limit. We test a variety of applications for assertions that fail only under weak isolation. MonkeyDB is able to break each of those assertions in a small number of attempts.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Theory of computation** → *Parallel computing models.*

Additional Key Words and Phrases: Applications of Storage Systems, Transactional Databases, Weak Isolation Levels, Testing

ACM Reference Format:
Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. 2021. MonkeyDB: Effectively Testing Correctness under Weak Isolation Levels. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 132 (October 2021), ?? pages. https://doi.org/10.1145/3485546

1 INTRODUCTION

Data storage is no longer about writing data to a single disk with a single point of access. Modern applications require not just data reliability, but also high-throughput concurrent accesses. Applications concerning supply chains, banking, etc. use traditional relational databases for storing and processing data, whereas applications such as social networking software and e-commerce platforms use cloud-based storage systems (such as Azure Cosmos DB [Paz 2018], Amazon DynamoDB [DeCandia et al. 2007], Facebook TAO [Bronson et al. 2013], etc.). We use the term *storage system* in this paper to refer to any such database system or service.

Authors' addresses: Ranadeep Biswas, Informal Systems, France, ranadeep@informal.systems; Diptanshu Kakwani, Microsoft, India, dkakwani@microsoft.com; Jyothi Vedurada, IIT Hyderabad, India, jyothiv@cse.iith.ac.in; Constantin Enea, IRIF, University of Paris & CNRS, France, cenea@irif.fr; Akash Lal, Microsoft Research, India, akashl@microsoft.com.

2475-1421/2021/10-ART132
https://doi.org/10.1145/3485546

```
// Append item to cart
AddItem(item i, userId) {
  begin()
  key = "cart:" + userId
  cart = read(key)
  cart.append(i)
  write(key, cart)
  commit()
}
```

```
// Fetch cart and delete item
DeleteItem(item i, userId) {
  begin()
  key = "cart:" + userId
  cart = read(key)
  cart.remove(i)
  write(key, cart)
  commit()
}
```

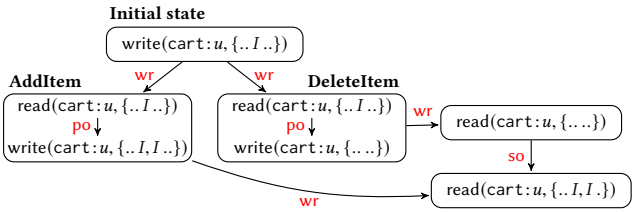


Fig. 1. A simple shopping cart service.

Providing high-throughput processing, unfortunately, comes at an unavoidable cost of weakening the guarantees offered to users. Concurrently-connected clients may end up observing different views of the same data. These "anomalies" can be prevented by using a strong *isolation level* such as *serializability* [Papadimitriou 1979], which essentially offers a single view of the data. However, serializability requires expensive synchronization and incurs a high performance cost. As a consequence, most storage systems use weaker isolation levels, such as *Causal Consistency* [Akkoorath and Bieniusa 2016; Lamport 1978; Lloyd et al. 2011], *Snapshot Isolation* [Berenson et al. 1995], *Read Committed* [Berenson et al. 1995], etc. for better performance. In a recent survey of database administrators [Pavlo 2017], 86% of the participants responded that most or all of the transactions in their databases execute at Read Committed isolation level.

A weaker isolation level allows for more possible behaviors than stronger isolation levels. It is up to the developers then to ensure that their application can tolerate this larger set of behaviors. Unfortunately, weak isolation levels are hard to understand or reason about [Adya 1999; Brutschy et al. 2017] and resulting application bugs can cause loss of business [Warszawski and Bailis 2017]. Consider a simple shopping cart application, inspired from Sivaramakrishnan et al. [2015], that stores a per-client shopping cart in a key-value store (*key* is the client ID and *value* is a multi-set of items). Figure 1 shows procedures for adding an item to the cart (`AddItem`) and deleting *all* instances of an item from the cart (`DeleteItem`). Each procedure executes in a transaction, represented by the calls to `begin` and `commit`. Suppose that initially, a user u has a single instance of item I in their cart. Then the user connects to the application via two different sessions (for instance, via two browser windows), adds I in one session (`AddItem(`I`, `u`)`) and deletes I in the other session (`DeleteItem(`I`, `u`)`). With serializability, the cart can either be left in the state $\{I\}$ (delete happened first, followed by the add) or $\emptyset$ (delete happened second). However, with Causal Consistency (or Read Committed), it is possible that with two sequential reads of the shopping cart, the cart is empty in the first read (signaling that the delete has succeeded), but there are *two* instances of I in the second read! Such anomalies, of deleted items reappearing, have been noted in previous work [DeCandia et al. 2007].

Testing storage-backed applications. This paper addresses the problem of *testing* code for correctness against weak behaviors: a developer should be able to write a test that runs their application and then asserts for correct behavior. The main difficulty with testing today is getting coverage of weak behaviors. Running against an actual production storage system is very likely to only result

in serializable behaviors because of their optimized implementation. For instance, only 0.0004% of all reads performed on Facebook's TAO storage system were not serializable [Lu et al. 2015]. Emulators, offered by cloud providers for local development, on the other hand, do not support weaker isolation levels at all [Microsoft 2020]. Another option, possible when the storage system is available open-source, is to set it up with a tool like Jepsen [Jepsen 2020] to inject noise (bring down replicas or delay packets on the network). This approach is unable to provide good coverage at the level of client operations [Rahmani et al. 2019] (more details in §8). Another line of work has focussed on finding anomalies by identifying non-serializable behavior (§10). Anomalies, however, do not always correspond to bugs [Brutschy et al. 2018; Gan et al. 2020]; they may either not be important (e.g., gathering statistics in a non-serializable fashion may not impact application correctness) or may already be handled by the application (e.g., checking and deleting duplicate items).

We present MonkeyDB, a mock in-memory storage system meant for testing correctness of storage-backed applications. MonkeyDB supports common APIs for accessing data (key-value updates, as well as SQL queries), making it an easy substitute for an actual storage system. MonkeyDB can be configured with one of several isolation levels. On a read operation, MonkeyDB computes the set of all possible return values allowed under the chosen isolation level, and randomly returns one of them. The developer can then simply execute their test multiple times to get coverage of possible weak behaviors. For the program in Figure 1, we can write a test asserting that two sequential reads cannot return empty-cart followed by $\{I, I\}$. Executing this test against MonkeyDB (with Read Committed) requires only 20 runs (on average) before the assertion fails. On the other hand, the test does not fail when using MySQL with Read Committed, even after 100K runs.

Testing, as opposed to static analysis or program verification, is still the most widely accepted method for ensuring correctness. MonkeyDB can work with any application without modification, neither does it require source access. A developer simply links their tests to run MonkeyDB instead of the production storage system. Our evaluation shows that using MonkeyDB, even with the simple strategy of randomly returning a valid read value, is able to break all invalid assertions, significantly out-performing any other testing solution.

Design of MonkeyDB. MonkeyDB does not rely on stress generation, fault injection, or data replication. Rather, it works directly with a formalization of the given isolation level in order to compute allowed return values.

The theory behind MonkeyDB builds on the axiomatic definitions of isolation levels introduced by Biswas and Enea [2019a]. These definitions use logical constraints (called *axioms*) to characterize the set of executions of a key-value store that conform to a particular isolation level (we discuss SQL queries later). These constraints refer to a specific set of relations between events/transactions in an execution that describe control-flow or data-flow dependencies: a program order po between events in the same transaction, a session order so between transactions in the same session[1], and a write-read wr (read-from) relation that associates each read event with a transaction that writes the value returned by the read. These relations along with the events (also called operations) in an execution are called a *history*. The history corresponding to the shopping cart anomaly explained above is given at the bottom of Figure 1. Read operations include the read value, and boxes group events from the same transaction. A history describes only the interaction with the key-value store, omitting application-side events (e.g., computing the value to be written to a key).

While the axiomatic formalization was already done in prior work, several enhancements were required to realize MonkeyDB. MonkeyDB implements a *centralized* operational semantics for key-value stores, which is based on these axiomatic definitions. Transactions are executed *serially*, one

[1]A session is a sequential interface to the storage system. It corresponds to what is also called a *connection*.

after another, the concurrency being simulated during the handling of read events. This semantics maintains a history that contains all the past events (from all transactions/sessions), and write events are simply added to the history. The value returned by a read event is established based on a non-deterministic choice of a write-read dependency (concerning this read event) that satisfies the axioms of the considered isolation level. Depending on the weakness of the isolation level, this makes it possible to return values written in arbitrarily "old" transactions, and simulate any concurrent behavior. For instance, the history in Figure 1 can be obtained by executing `AddItem`, `DeleteItem`, and then the two reads (serially). The read in `DeleteItem` can take its value from the initial state and "ignore" the previously executed `AddItem`, because the obtained history validates the axioms of Causal Consistency (or Read Committed). The same happens for the two later reads in the same session, the first one being able to read from `DeleteItem` and the second one from `AddItem`. We also preserve commit-order constraints across read operations in order to *incrementally* deal with a live history.

We formally prove that this semantics does indeed simulate any concurrent behavior, by showing that it is equivalent to a semantics where transactions are allowed to interleave. In comparison with concrete implementations, this semantics makes it possible to handle a wide range of isolation levels in a uniform way. It only has two sources of non-determinism: the order in which entire transactions are submitted, and the choice of write-read dependencies in read events. This enables better coverage of possible behaviors, the penalty in performance not being an issue in safety testing workloads which are usually small (see our evaluation).

We also extend our semantics to cover SQL queries as well, by compiling SQL queries down to transactions with multiple key-value reads/writes. A table in a relational database is represented using a set of primary key values (identifying uniquely the set of rows) and a set of keys, one for each cell in the table. The set of primary key values is represented using a set of Boolean key-value pairs that simulate its characteristic function (adding or removing an element corresponds to updating one of these keys to true or false). Then, SQL queries are compiled to read or write accesses to the keys representing a table. For instance, a `SELECT` query that retrieves the set of rows in a table that satisfy a `WHERE` condition is compiled to (1) reading Boolean keys to identify the primary key values of the rows contained in the table, (2) reading keys that represent columns used in the `WHERE` condition, and (3) reading all the keys that represent cells in a row satisfying the `WHERE` condition. This rewriting contains the minimal set of accesses to the cells of a table that are needed to ensure the conventional specification of SQL. It makes it possible to "export" formalizations of key-value store isolation levels to SQL transactions.

Contributions. This paper makes the following contributions:

- We define an operational semantics, based on the axiomatic definitions introduced by Biswas and Enea [2019a], for key-value stores under various isolation levels. We show that our semantics simulates all concurrent behaviors with executions where transactions execute serially (§4).
- We broaden the scope of the key-value store semantics to SQL transactions using a compiler that rewrites SQL queries to key-value accesses (§5),
- The operational semantics and the SQL compiler are implemented in a tool called MonkeyDB (§6). It randomly resolves possible choices to provide coverage of weak behaviors. It supports both a key-value interface as well as SQL, making it readily compatible with any storage-backed application.
- We present an evaluation of MonkeyDB on several applications, including a series of micro-benchmarks inspired from real applications (§7), as well as the well-known OLTPBench [Difallah et al. 2013] that is the standard for evaluating databases on *online transaction*

$$k \in \text{Keys} \quad x \in \text{Vars} \quad tab \in \mathbb{T} \quad \vec{c}, \vec{c_1}, \vec{c_2} \in \mathbb{C}^*$$

$$\begin{aligned}
\text{Prog} &::= \text{Sess} \mid \text{Sess} \,||\, \text{Prog} \\
\text{Sess} &::= \text{Trans} \mid \text{Trans}; \text{Sess} \\
\text{Trans} &::= \texttt{begin}; \text{Body}; \texttt{commit} \\
\text{Body} &::= \text{Instr} \mid \text{Instr}; \text{Body} \\
\text{Instr} &::= \text{InstrKV} \mid \text{InstrSQL} \mid x := e \mid \texttt{if}(\phi(\vec{x}))\{\text{Instr}\} \\
\text{InstrKV} &::= x := \texttt{read}(k) \mid \texttt{write}(k, x) \\
\text{InstrSQL} &::= \texttt{SELECT } \vec{c_1} \texttt{ AS } x \texttt{ FROM } tab \texttt{ WHERE } \phi(\vec{c_2}) \mid \\
&\quad\ \texttt{INSERT INTO } tab \texttt{ VALUES } \vec{x} \mid \\
&\quad\ \texttt{DELETE FROM } tab \texttt{ WHERE } \phi(\vec{c}) \mid \\
&\quad\ \texttt{UPDATE } tab \texttt{ SET } \vec{c_1} = \vec{x} \texttt{ WHERE } \phi(\vec{c_2})
\end{aligned}$$

Fig. 2. Program syntax. The set of all keys is denoted by Keys, Vars denotes the set of local variables, $\mathbb{T}$ the set of table names, and $\mathbb{C}$ the set of column names. We use ϕ to denote Boolean expressions, and e to denote expressions interpreted as values. We use $\vec{\cdot}$ to denote vectors of elements.

processing (OLTP) workloads (§8). MonkeyDB provides a high coverage of weak behaviors and is able to break all invalid assertions in a few attempts, significantly better than prior testing solutions.

MonkeyDB and the source code of our benchmarks are available at [Biswas et al. 2021b]. An extended version of the paper is available at [Biswas et al. 2021a].

2 PROGRAMMING LANGUAGE

Figure 2 lists the definition of two simple programming languages that we use to represent applications running on top of key-value or SQL stores, respectively. A program is a set of *sessions* running in parallel, each session being composed of a sequence of *transactions*. Each transaction is delimited by `begin` and `commit` instructions, and its body contains instructions that access the store and manipulate a set of local variables Vars.[2] We use symbols x, y, etc. to denote elements of Vars.

In case of a program running on top of a key-value store, the instructions can be: reading the value of a key and storing it to a local variable x ($x := \texttt{read}(k)$) , writing the value of a local variable x to a key ($\texttt{write}(k, x)$), or an assignment to a local variable x. The set of values of keys or local variables is denoted by Vals. Assignments to local variables use expressions interpreted as values whose syntax is left unspecified. Each of these instructions can be guarded by a Boolean condition $\phi(\vec{x})$ over a set of local variables $\vec{x}$ (their syntax is not important). Other constructs like `while` loops can be defined in a similar way. Let $\mathcal{P}_{KV}$ denote the set of programs where a transaction body can contain only such instructions.

For programs running on top of SQL stores, the instructions include simplified versions of standard SQL instructions and assignments to local variables. These programs run in the context of a *database schema* which is a (partial) function $\mathcal{S} : \mathbb{T} \rightharpoonup 2^{\mathbb{C}}$ mapping table names in $\mathbb{T}$ to sets of column names in $\mathbb{C}$. The SQL store is an *instance* of a database schema $\mathcal{S}$, i.e., a function $\mathcal{D} : \text{dom}(\mathcal{S}) \rightarrow 2^{\mathbb{R}}$ mapping each table tab in the domain of $\mathcal{S}$ to a set of *rows* of tab, i.e., functions

[2]For simplicity, we assume that all the transactions in the program commit. Aborted transactions can be ignored when reasoning about safety because their effects should be invisible to other transactions.

$r : \mathcal{S}(tab) \to \text{Vals}$. We use $\mathbb{R}$ to denote the set of all rows. The SELECT instruction retrieves the columns $\vec{c_1}$ from the set of rows of *tab* that satisfy $\phi(\vec{c_2})$ ($\vec{c_2}$ denotes the set of columns used in this Boolean expression), and stores them into a variable x. INSERT adds a new row to *tab* with values $\vec{x}$, and DELETE deletes all rows from *tab* that satisfy a condition $\phi(\vec{c})$. The UPDATE instruction assigns the columns $\vec{c_1}$ of all rows of *tab* that satisfy $\phi(\vec{c_2})$ with values in $\vec{x}$. Let $\mathcal{P}_{SQL}$ denote the set of programs where a transaction body can contain only such instructions.

3 ISOLATION LEVELS FOR KEY-VALUE STORES

We present the axiomatic framework introduced by Biswas and Enea [2019a] for defining isolation levels in key-value stores.[3] Isolation levels are defined as logical constraints, called *axioms*, over *histories*, which are an abstract representation of the interaction between a program and the store in a concrete execution.

3.1 Histories

Programs interact with a key-value store by issuing transactions formed of read and write instructions. The effect of executing one such instruction is represented using an *operation*, which is an element of the set

$$\text{Op} = \{\text{read}_i(k, v), \text{write}_i(k, v) : i \in \text{OpId}, k \in \text{Keys}, v \in \text{Vals}\}$$

where $\text{read}_i(k, v)$ (resp., $\text{write}_i(k, v)$) corresponds to reading a value v from a key k (resp., writing v to k). Each operation is associated with an identifier i from an arbitrary set OpId. We omit operation identifiers when they are not important.

Definition 3.1. A *transaction log* $\langle t, O, \text{po}\rangle$ is a transaction identifier t and a finite set of operations O along with a strict total order po on O, called *program order*.

The program order po represents the order between instructions in the body of a transaction. We assume that each transaction log is well-formed in the sense that if a read of a key k is preceded by a write to k in po, then it should return the value written by the last write to k before the read (w.r.t. po). This property is implicit in the definition of every isolation level that we are aware of. For simplicity, we may use the term *transaction* instead of transaction log. The set of all transaction logs is denoted by Tlogs.

The set of read operations $\text{read}(k, \_)$ in a transaction log t that are *not* preceded by a write to k in po is denoted by $\text{reads}(t)$. As mentioned above, the other read operations take their values from writes in the same transaction and their behavior is independent of other transactions. Also, the set of write operations $\text{write}(k, \_)$ in t that are *not* followed by other writes to k in po is denoted by $\text{writes}(t)$. If a transaction contains multiple writes to the same key, then only the last one (w.r.t. po) can be visible to other transactions (w.r.t. any isolation level that we are aware of). The extension to sets of transaction logs is defined as usual. Also, we say that a transaction log t *writes* a key k, denoted by t writes k, when $\text{write}_i(k, v) \in \text{writes}(t)$ for some i and v.

A *history* contains a set of transaction logs (with distinct identifiers) ordered by a (partial) *session order* so that represents the order between transactions in the same session.[4] It also includes a *write-read* relation (also called read-from) that "justifies" read values by associating each read to a transaction that wrote the value returned by the read.

[3] Isolation levels are called consistency models by Biswas and Enea [2019a].
[4] In the context of our programming language, so would be a union of total orders. This constraint is not important for defining isolation levels.

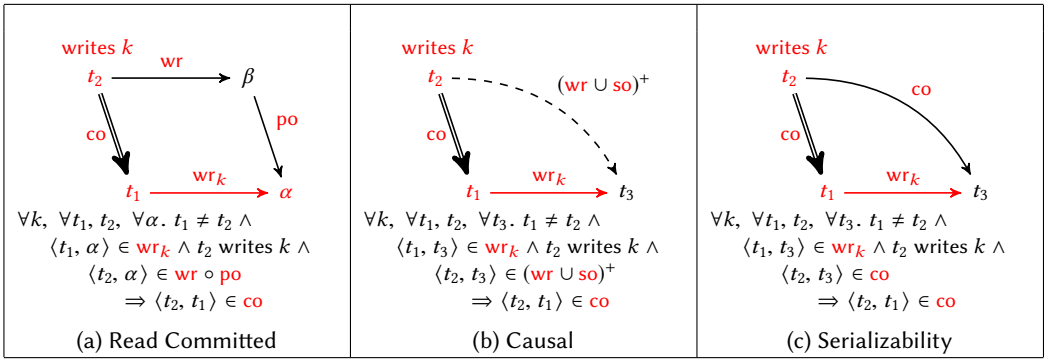


Fig. 3. Axioms defining isolations levels. The reflexive and transitive, resp., transitive, closure of a relation rel is denoted by rel^*, resp., rel^+. Also, $\circ$ denotes the composition of two relations, i.e., $rel_1 \circ rel_2 = \{\langle a, b\rangle | \exists c. \langle a, c\rangle \in rel_1 \wedge \langle c, b\rangle \in rel_2\}$.

Definition 3.2. A *history* $\langle T, \mathrm{so}, \mathrm{wr}\rangle$ is a set of transaction logs T along with a strict partial *session order* so, and a *write-read* relation $\mathrm{wr} \subseteq T \times \mathrm{reads}(T)$ such that the inverse of wr is a total function, and if $(t, \mathrm{read}(k, v)) \in \mathrm{wr}$, then $\mathrm{write}(k, v) \in t$, and $\mathrm{so} \cup \mathrm{wr}$ is acyclic.

To simplify the technical exposition, we assume that every history includes a distinguished transaction log writing the initial values of all keys. This transaction log precedes all the other transaction logs in so. We use $h, h_1, h_2, \ldots$ to range over histories. The set of transaction logs T in a history $h = \langle T, \mathrm{so}, \mathrm{wr}\rangle$ is denoted by $\mathrm{TLogs}(h)$.

For a key k, wr_k denotes the restriction of wr to reads of k, i.e., , $\mathrm{wr}_k = \mathrm{wr} \cap (T \times \{\mathrm{read}(k, v) \mid v \in \mathrm{Vals}\})$. Moreover, we extend the relations wr and wr_k to pairs of transactions by $\langle t_1, t_2\rangle \in \mathrm{wr}$, resp., $\langle t_1, t_2\rangle \in \mathrm{wr}_k$, iff there exists a read operation $\mathrm{read}(k, v) \in \mathrm{reads}(t_2)$ such that $\langle t_1, \mathrm{read}(k, v)\rangle \in \mathrm{wr}$, resp., $\langle t_1, \mathrm{read}(k, v)\rangle \in \mathrm{wr}_k$. We say that the transaction log t_1 is *read* by the transaction log t_2 when $\langle t_1, t_2\rangle \in \mathrm{wr}$.

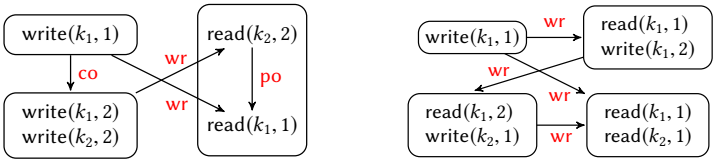
3.2 Axiomatic Framework

A history is said to satisfy a certain isolation level if there exists a strict total order co on its transaction logs, called *commit order*, which extends the write-read relation and the session order, and which satisfies certain properties. These properties, called *axioms*, relate the commit order with the session-order and the write-read relation in the history. They are defined as first-order formulas of the following form:

$$\forall k,\ \forall t_1 \neq t_2,\ \forall \tau.\\ \langle t_1, \tau\rangle \in \mathrm{wr}_k \wedge t_2 \text{ writes } k \wedge \phi(t_2, \tau) \Rightarrow \langle t_2, t_1\rangle \in \mathrm{co} \quad (1)$$

where ϕ is a property relating t_2 and τ (i.e., the read or the transaction reading from t_1) that varies from one axiom to another.[5] Intuitively, this axiom schema states the following: in order for τ to read specifically t_1's write on k, it must be the case that every t_2 that also writes k and satisfies $\phi(t_2, \tau)$ was committed before t_1. The property ϕ relates t_2 and τ using the relations in a history and the commit order. Figure 3 shows the axioms defining three isolation levels: Read Committed, Causal Consistency, and Serializability (see Biswas and Enea [2019a] for axioms defining Read Atomic, Prefix, and Snapshot Isolation).

[5]These formulas are interpreted on tuples $\langle h, \mathrm{co}\rangle$ of a history h and a commit order co on the transactions in h as usual.



(a) Read Committed violation. (b) Causal violation.

Fig. 4. Histories used to explain the axioms in Figure 3.

For instance, Read Committed [Berenson et al. 1995] requires that every read returns a value written in a committed transaction, and also, that the reads in the same transaction are "monotonic", i.e., they do not return values that are older, w.r.t. the commit order, than values read in the past. While the first condition holds for every history (because of the surjectivity of wr), the second condition is expressed by the axiom Read Committed in Figure 3a, which states that for any transaction t_1 writing a key k that is read at an operaion α in a transaction, the set of transactions t_2 writing k and read previously in the same transaction (these reads may concern other keys) must precede t_1 in commit order. For instance, Figure 4a shows a history and a (partial) commit order that does not satisfy this axiom because read$(k_1, 1)$ returns the value written in a transaction "older" than the transaction read in the previous read$(k_2, 2)$.

The axiom defining Causal Consistency [Lamport 1978] states that for any transaction t_1 writing a key k that is read in a transaction t_3, the set of $(\mathrm{wr} \cup \mathrm{so})^+$ predecessors of t_3 writing k must precede t_1 in commit order ($(\mathrm{wr} \cup \mathrm{so})^+$ is usually called the *causal* order). A violation of this axiom can be found in Figure 4b: the transaction t_2 writing 2 to k_1 is a $(\mathrm{wr} \cup \mathrm{so})^+$ predecessor of the transaction t_3 reading 1 from k_1 because the transaction t_4, writing 1 to k_2, reads k_1 from t_2 and t_3 reads k_2 from t_4. This implies that t_2 should precede in commit order the transaction t_1 writing 1 to k_1, which again, is inconsistent with the write-read relation (t_2 reads from t_1).

Finally, Serializability [Papadimitriou 1979] requires that for any transaction t_1 writing to a key k that is read in a transaction t_3, the set of co predecessors of t_3 writing k must precede t_1 in commit order. This ensures that each transaction observes the effects of all the co predecessors.

Definition 3.3. For an isolation level I defined by a set of axioms X, a history $h = \langle T, \mathrm{so}, \mathrm{wr} \rangle$ *satisfies* I iff there is a strict total order co s.t. $\mathrm{wr} \cup \mathrm{so} \subseteq \mathrm{co}$ and $\langle h, \mathrm{co} \rangle$ satisfies X.[6]

4 OPERATIONAL SEMANTICS FOR $\mathcal{P}_{KV}$

We define a small-step operational semantics for key-value store programs, which is parametrized by an isolation level I. Transactions are executed *atomically* (without interruption) one after another, and the values returned by read operations are decided using the axiomatic definition of I. The semantics maintains a history of previously executed operations, and the value returned by a read is chosen non-deterministically as long as extending the current history with the corresponding write-read dependency satisfies the axioms of I. We show that this semantics is sound and complete for any natural isolation level I, i.e., it generates precisely the same set of histories as a *baseline* semantics that allows for the fine-grain interleaving of operations in different transactions, and arbitrary values for read operations as long as they can be proved to be correct at the end of the execution.

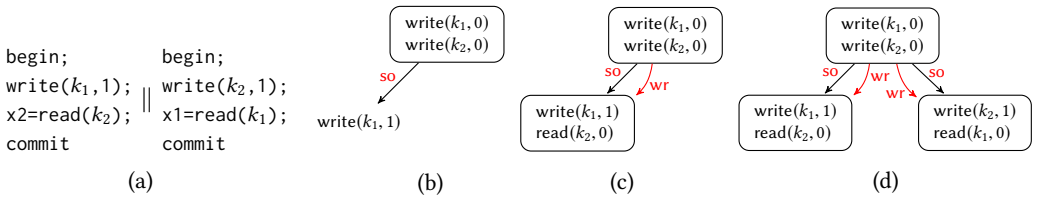


Fig. 5. The Causal semantics of the program in (a), assuming that the transaction on the left is scheduled first.

4.1 Definition of the Operational Semantics

We use the program in Figure 5a to give an overview of our semantics, assuming Causal Consistency. This program has two concurrent transactions whose reads can both return the initial value 0, which is not possible under Serializability.

Our semantics executes transactions in their entirety one after another (without interleaving operations from different transactions), maintaining a history that contains all the executed operations. We assume that the transaction on the left executes first. Initially, the history contains a fictitious transaction log that writes the initial value 0 to all the keys, and that will precede all the transaction logs created during the execution in session order.

Executing a write instruction consists in simply appending the corresponding write operation to the log of the current transaction. For instance, executing the first write (and `begin`) in our example results in adding a transaction log that contains a write operation (see Figure 5b). The execution continues with the read instruction from the same transaction, and it cannot switch to the other transaction.

The execution of a read instruction consists in choosing non-deterministically a write-read dependency that validates Causal when added to the current history. In our example, executing `read`(k_2) results in adding a write-read dependency from the transaction log writing initial values, which determines the return value of the `read` (see Figure 5c). This choice makes the obtained history satisfy Causal.

The second transaction executes in a similar manner. When executing its read instruction, the chosen write-read dependency is again related to the transaction log writing initial values (see Figure 5d). This choice is valid under Causal. Since it is possible that a read does not return a value written in the preceding transaction, this semantics is able to simulate all the "anomalies" of a weak isolation level (this execution being an example).

Formally, the operational semantics is defined as a transition relation $\Rightarrow_I$ between *configurations*, which are defined as tuples containing the following:

- history h storing the operations executed in the past,
- identifier j of the current session,
- local variable valuation γ for the current transaction,
- code B that remains to be executed from the current transaction, and
- sessions/transactions P that remain to be executed from the original program.

For readability, we define a program as a partial function $\mathsf{P} : \mathsf{SessId} \rightharpoonup \mathsf{Sess}$ that associates session identifiers in SessId with concrete code as defined in Figure 2 (i.e., sequences of transactions). Similarly, the session order so in a history is defined as a partial function $\mathsf{so} : \mathsf{SessId} \rightharpoonup \mathsf{Tlogs}^*$ that associates session identifiers with sequences of transaction logs. Two transaction logs are ordered by so if one occurs before the other in some sequence $\mathsf{so}(j)$ with $j \in \mathsf{SessId}$.

[6]Isolation levels like Snapshot Isolation require more than one axiom.

$$\textsc{spawn}\ \frac{t \text{ fresh} \quad \mathsf{P}(j) = \mathtt{begin}; \mathsf{Body}; \mathtt{commit}; \mathsf{S}}{h, \_, \_, \epsilon, \mathsf{P} \Rightarrow_I h \oplus_j \langle t, \emptyset, \emptyset\rangle, j, \emptyset, \mathsf{Body}, \mathsf{P}[j \mapsto \mathsf{S}]}$$

$$\textsc{if-true}\ \frac{\psi(\vec{x})[x \mapsto \gamma(x) : x \in \vec{x}]\ \mathsf{true}}{h, j, \gamma, \mathtt{if}(\psi(\vec{x}))\{\mathsf{Instr}\}; \mathsf{B}, \mathsf{P} \Rightarrow_I h, j, \gamma, \mathsf{Instr}; \mathsf{B}, \mathsf{P}}$$

$$\textsc{if-false}\ \frac{\psi(\vec{x})[x \mapsto \gamma(x) : x \in \vec{x}]\ \mathsf{false}}{h, j, \gamma, \mathtt{if}(\psi(\vec{x}))\{\mathsf{Instr}\}; \mathsf{B}, \mathsf{P} \Rightarrow_I h, j, \gamma, \mathsf{B}, \mathsf{P}}$$

$$\textsc{write}\ \frac{v = \gamma(x) \quad i \text{ fresh}}{h, j, \gamma, \mathtt{write}(k, x); \mathsf{B}, \mathsf{P} \Rightarrow_I h \oplus_j \mathsf{write}_i(k, v), j, \gamma, \mathsf{B}, \mathsf{P}}$$

$$\textsc{read-local}\ \frac{\mathsf{write}(k, v) \text{ is the last write on } k \text{ in } t \text{ w.r.t. } \mathrm{po} \qquad i \text{ fresh}}{h, j, \gamma, x := \mathtt{read}(k); \mathsf{B}, \mathsf{P} \Rightarrow_I h \oplus_j \mathsf{read}_i(k, v), j, \gamma[x \mapsto v], \mathsf{B}, \mathsf{P}}$$

$$\textsc{read-extern}\ \frac{\begin{array}{c} h = (T, \mathrm{so}, \mathrm{wr}) \\ t \text{ is the id of the last transaction log in } \mathrm{so}(j) \qquad \mathsf{write}(k, v) \in \mathsf{writes}(t') \text{ with } t' \in T \text{ and } t' \neq t \\ i \text{ fresh} \qquad h' = (h \oplus_j \mathsf{read}_i(k, v)) \oplus \mathrm{wr}(t', \mathsf{read}_i(k, v)) \qquad h' \text{ satisfies } I \end{array}}{h, j, \gamma, x := \mathtt{read}(k); \mathsf{B}, \mathsf{P} \Rightarrow_I h', j, \gamma[x \mapsto v], \mathsf{B}, \mathsf{P}}$$

Fig. 6. Operational semantics for $\mathcal{P}_{KV}$ programs under isolation level I. For a function $f : A \rightharpoonup B$, $f[a \mapsto b]$ denotes the function $f' : A \rightharpoonup B$ defined by $f'(c) = f(c)$, for every $c \neq a$ in the domain of f, and $f'(a) = b$.

Before presenting the definition of $\Rightarrow_I$, we introduce some notation. Let h be a history that contains a representation of so as above. We use $h \oplus_j \langle t, O, \mathrm{po}\rangle$ to denote a history where $\langle t, O, \mathrm{po}\rangle$ is appended to $\mathrm{so}(j)$. Also, for an operation o, $h \oplus_j o$ is the history obtained from h by adding o to the last transaction log in $\mathrm{so}(j)$ and as a last operation in the program order of this log (i.e., if $\mathrm{so}(j) = \sigma; \langle t, O, \mathrm{po}\rangle$, then the session order so' of $h \oplus_j o$ is defined by $\mathrm{so}'(k) = \mathrm{so}(k)$ for all $k \neq j$ and $\mathrm{so}(j) = \sigma; \langle t, O \cup o, \mathrm{po} \cup \{(o', o) : o' \in O\}\rangle$). Finally, for a history $h = \langle T, \mathrm{so}, \mathrm{wr}\rangle$, $h \oplus \mathrm{wr}(t, o)$ is the history obtained from h by adding (t, o) to the write-read relation.

Figure 6 lists the rules defining $\Rightarrow_I$. The SPAWN rule starts a new transaction, provided that there is no other live transaction ($\mathsf{B} = \epsilon$). It adds an empty transaction log to the history and schedules the body of the transaction. IF-TRUE and IF-FALSE check the truth value of a Boolean condition of an `if` conditional. WRITE corresponds to a write instruction and consists in simply adding a write operation to the current history. READ-LOCAL and READ-EXTERN concern read instructions. READ-LOCAL handles the case where the read follows a write on the same key k in the same transaction: the read returns the value written by the last write on k in the current transaction. Otherwise, READ-EXTERN corresponds to reading a value written in another transaction t' (t is the id of the log of the current transaction). The transaction t' is chosen non-deterministically as long as extending the current history with the write-read dependency associated to this choice leads to a history that still satisfies I. READ-EXTERN applies only when the current transaction contains no write to the same key.

An *initial* configuration for program P contains the program P along with a history $h = \langle\{t_0\}, \emptyset, \emptyset\rangle$, where t_0 is a transaction log containing only writes that write the initial values of all keys, and empty current transaction code ($\mathsf{B} = \epsilon$). An execution of a program P under an isolation level I is a sequence of configurations $c_0 c_1 \ldots c_n$ where c_0 is an initial configuration for P, and $c_m \Rightarrow_I c_{m+1}$, for every $0 \leq m < n$. We say that c_n is *I-reachable* from c_0. The history of such an execution is the history h in the last configuration c_n. A configuration is called *final* if it contains the empty program ($\mathsf{P} = \emptyset$). Let $\mathrm{hist}_I(\mathsf{P})$ denote the set of all histories of an execution of P under I that ends in a final configuration.

$$\textsc{spawn}^*\ \frac{t \text{ fresh} \quad \mathsf{P}(j) = \mathtt{begin}; \mathsf{Body}; \mathtt{commit}; \mathsf{S} \quad \vec{\mathsf{B}}(j) = \epsilon}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow h \oplus_j \langle t, \emptyset, \emptyset\rangle, \vec{\gamma}[j \mapsto \emptyset], \vec{\mathsf{B}}[j \mapsto \mathsf{Body}], \mathsf{P}[j \mapsto \mathsf{S}]}$$

$$\textsc{if-true}^*\ \frac{\psi(\vec{x})[x \mapsto \vec{\gamma}(j)(x) : x \in \vec{x}] \text{ true} \qquad \vec{\mathsf{B}}(j) = \mathtt{if}(\psi(\vec{x}))\{\mathsf{Instr}\}; \mathsf{B}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow h, \vec{\gamma}, \vec{\mathsf{B}}[j \mapsto \mathsf{Instr}; \mathsf{B}], \mathsf{P}}$$

$$\textsc{if-false}^*\ \frac{\psi(\vec{x})[x \mapsto \vec{\gamma}(j)(x) : x \in \vec{x}] \text{ false} \qquad \vec{\mathsf{B}}(j) = \mathtt{if}(\psi(\vec{x}))\{\mathsf{Instr}\}; \mathsf{B}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow h, \vec{\gamma}, \vec{\mathsf{B}}[j \mapsto \mathsf{B}], \mathsf{P}}$$

$$\textsc{write}^*\ \frac{v = \vec{\gamma}(j)(x) \quad i \text{ fresh} \quad \vec{\mathsf{B}}(j) = \mathtt{write}(k, x); \mathsf{B}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow h \oplus_j \mathsf{write}_i(k, v), \vec{\gamma}, \vec{\mathsf{B}}[j \mapsto \mathsf{B}], \mathsf{P}}$$

$$\textsc{read-local}^*\ \frac{\mathsf{write}(k, v) \text{ is the last write on } k \text{ in } t \qquad i \text{ fresh} \qquad \vec{\mathsf{B}}(j) = x := \mathtt{read}(k); \mathsf{B}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow h \oplus_j \mathsf{read}_i(k, v), \vec{\gamma}[(j, x) \mapsto v], \vec{\mathsf{B}}[j \mapsto \mathsf{B}], \mathsf{P}}$$

$$\textsc{read-extern}^*\ \frac{\begin{array}{c}\vec{\mathsf{B}}(j) = x := \mathtt{read}(k); \mathsf{B} \qquad h = (T, \mathsf{so}, \mathsf{wr}) \qquad t \text{ is the id of the last transaction log in } \mathsf{so}(j) \\ \mathsf{write}(k, v) \in \mathsf{writes}(t') \text{ with } t' \in \mathsf{compTrans}(h, \vec{\mathsf{B}}) \text{ and } t \neq t' \\ i \text{ fresh} \qquad h' = (h \oplus_j \mathsf{read}_i(k, v)) \oplus \mathsf{wr}(t', \mathsf{read}_i(k, v))\end{array}}{h, \vec{\gamma}, \vec{\mathsf{B}}, \mathsf{P} \Rightarrow h', \vec{\gamma}[(j, x) \mapsto v], \vec{\mathsf{B}}[j \mapsto \mathsf{B}], \mathsf{P}}$$

Fig. 7. A baseline operational semantics for $\mathcal{P}_{KV}$ programs. Above, $\mathsf{compTrans}(h, \vec{\mathsf{B}})$ denotes the set of transaction logs in h that excludes those corresponding to live transactions, i.e., transaction logs $t'' \in T$ such that t'' is the last transaction log in some $\mathsf{so}(j')$ and $\vec{B}(j') \neq \epsilon$.

4.2 Correctness of the Operational Semantics

We define the correctness of $\Rightarrow_I$ in relation to a *baseline* semantics where operations from different transactions can interleave arbitrarily, and the values returned by read operations are only constrained to come from committed transactions. This semantics is represented by a transition relation $\Rightarrow$, which is defined by a set of rules that are analogous to $\Rightarrow_I$. Since it allows transactions to interleave, a configuration contains a history h, the sessions/transactions P that remain to be executed, and:

- a valuation map $\vec{\gamma}$ that records local variable values in the current transaction of each session ($\vec{\gamma}$ associates identifiers of sessions that have live transactions with valuations of local variables),
- a map $\vec{B}$ that stores the code of each live transaction (associating session identifiers with code).

Figure 7 lists the rules defining $\Rightarrow$. SPAWN* starts a new transaction in a session j provided that this session has no live transaction ($\vec{\mathsf{B}}(j) = \epsilon$). Compared to SPAWN in Figure 6, this rule allows unfinished transactions in other sessions. READ-EXTERN* does not check conformance to I, but it allows a read to only return a value written in a completed (committed) transaction. In this work, we consider only isolation levels satisfying this constraint. The rest of the rules are similar to those

defining $\Rightarrow_I$. Executions, initial and final configurations are defined as in the case of $\Rightarrow_I$. The history of an execution is still defined as the history in the last configuration. Let $\mathsf{hist}_*(\mathsf{P})$ denote the set of all histories of an execution of P w.r.t. $\Rightarrow$ that ends in a final configuration.

Practical isolation levels satisfy a "prefix-closure" property saying that if the axioms of I are satisfied by a pair $\langle h_2, \mathrm{co}_2\rangle$, then they are also satisfied by every *prefix* of $\langle h_2, \mathrm{co}_2\rangle$. A prefix of $\langle h_2, \mathrm{co}_2\rangle$ contains a prefix of the sequence of transactions in h_2 when ordered according to co_2, and the last transaction log in this prefix is possibly incomplete. In general, this prefix-closure property holds for isolation levels I that are defined by axioms as in (1), provided that the property $\phi(t_2, \alpha)$ is *monotonic*, i.e., the set of models in the context of a pair $\langle h_2, \mathrm{co}_2\rangle$ is a *superset* of the set of models in the context of a prefix $\langle h_1, \mathrm{co}_1\rangle$ of $\langle h_2, \mathrm{co}_2\rangle$. For instance, the property ϕ in the axiom defining Causal is $(t_2, \alpha) \in (\mathrm{wr} \cup \mathrm{so})^+$, which is clearly monotonic. In general, standard isolation levels are defined using a property α of the form $(t_2, \alpha) \in R$ where R is an expression built from the relations po, so, wr, and co using (reflexive and) transitive closure and composition of relations [Biswas and Enea 2019a]. Such properties are monotonic in general (they would not be if those expressions would use the negation/complement of a relation). An axiom as in (1) is called *monotonic* when the property ϕ is monotonic.

Formally, a prefix of a tuple $\{h_2, \mathrm{co}_2\}$ is defined as follows. For a relation $R \subseteq A \times B$, the restriction of R to $A' \times B'$, denoted by $R \downarrow A' \times B'$, is defined by $\{(a, b) : (a, b) \in R, a \in A', b \in B'\}$. For $h_1 = \langle T_1, \mathrm{so}_1, \mathrm{wr}_1\rangle$ and $h_2 = \langle T_2, \mathrm{so}_2, \mathrm{wr}_2\rangle$, we say that $\langle h_1, \mathrm{co}_1\rangle$ is a *prefix* of $\langle h_2, \mathrm{co}_2\rangle$, denoted by $\langle h_1, \mathrm{co}_1\rangle \leq \langle h_2, \mathrm{co}_2\rangle$, when $T_1 = T_1' \cup \{\langle t, O, \mathrm{po}\rangle\}$, $T_2 = T_2' \cup \{\langle t, O', \mathrm{po}'\rangle\}$, $T_1' \subseteq T_2'$, $O \subseteq O'$, $\mathrm{po} = \mathrm{po}' \downarrow O \times O$, $\mathrm{so}_1 = \mathrm{so}_2 \downarrow T_1 \times T_1$, $\mathrm{wr}_1 = \mathrm{wr}_2 \downarrow T_1 \times \mathrm{reads}(T_1)$, and $\mathrm{co}_1 = \mathrm{co}_2 \downarrow T_1 \times T_1$.

Then, a property $\phi(t_2, \alpha)$ used to define an axiom like in (1), is called *monotonic* iff for every $\langle h_1, \mathrm{co}_1\rangle \leq \langle h_2, \mathrm{co}_2\rangle$,

$$\forall t_2, \forall \alpha. \langle h_2, \mathrm{co}_2\rangle \models \phi(t_2, \alpha) \Rightarrow \langle h_1, \mathrm{co}_1\rangle \models \phi(t_2, \alpha).$$

Lemma 4.1. *For any monotonic axiom X, if $\langle h_1, \mathrm{co}_1\rangle \leq \langle h_2, \mathrm{co}_2\rangle$, then*

$$\langle h_2, \mathrm{co}_2\rangle \textit{ satisfies } X \Rightarrow \langle h_1, \mathrm{co}_1\rangle \textit{ satisfies } X$$

Proof. (Sketch) Given a monotonic axiom, the number of instantiations of $\forall k$, $\forall t_1$, $\forall t_2$, and $\forall \alpha$ from (1) that satisfy the left-hand side of the entailment in the context of $\langle h_1, \mathrm{co}_1\rangle$ is a subset of the same type of instantiations in the context of $\langle h_2, \mathrm{co}_2\rangle$. Therefore, the co constraints imposed in the context of $\langle h_1, \mathrm{co}_1\rangle$ (by the right-hand side of the entailment) are a subset of the co constraints imposed in the context of $\langle h_2, \mathrm{co}_2\rangle$. Since the latter are satisfied (because $\langle h_2, \mathrm{co}_2\rangle$ satisfies X), the former are also satisfied and hence, $\langle h_1, \mathrm{co}_1\rangle$ satisfies X. □

Lemma 4.1 extends obviously to isolation levels defined as conjunctions of axioms (which is the case for all the isolation levels that we are aware of [Biswas and Enea 2019a]).

The following theorem shows that $\mathsf{hist}_I(\mathsf{P})$ is precisely the set of histories under the baseline semantics, which satisfy I (the validity of the reads is checked at the end of an execution), provided that the axioms of I are monotonic. The $\subseteq$ direction follows mostly from the fact that $\Rightarrow_I$ is more constrained than $\Rightarrow$. For the opposite direction, given a history h that satisfies I, i.e., there exists a commit order co such that $\langle h, \mathrm{co}\rangle$ satisfies the axioms of I, we can show that there exists an execution under $\Rightarrow_I$ with history h, where transactions execute atomically in the order defined by co. The prefix closure property is used to prove that READ-EXTERN transitions are enabled (these transitions get executed with a prefix of h).

Theorem 4.2. *For any isolation level I defined by a set of monotonic axioms,* $\mathsf{hist}_I(\mathsf{P}) = \{h \in \mathsf{hist}_*(\mathsf{P}) : h \textit{ satisfies } I\}$.

Proof. (Sketch) For the direction $\subseteq$, let $c_0 c_1 \ldots c_n$ be an execution under $\Rightarrow_I$, where c_n is a final configuration. We need to show that the history h_n contained in c_n belongs to $\mathsf{hist}_*(\mathrm{P})$ and that it satisfies I. The fact that $h_n \in \mathsf{hist}_*(\mathrm{P})$ is a direct consequence of the fact that $\Rightarrow_I$ is more constrained than $\Rightarrow$. To prove that h_n satisfies I, let c_j be the latest configuration in the execution that is obtained from c_{j-1} through an application of read-extern. By the definition of this rule, the history h_j in c_j satisfies I. Since the write-read relation of h_j is identical to that of h_n, any axiom of the form (1) satisfied by h_j is also satisfied by h_n (the set of instantiations of $\forall t_1$ and $\forall \alpha$ in (1) that satisfy the left part of the entailment are the same in h_j and h_n). Therefore, h_n satisfies I, which concludes this part of the proof.

For the reverse, let $h = \langle T, \mathsf{so}, \mathsf{wr} \rangle \in \mathsf{hist}_*(\mathrm{P})$ that satisfies I. Since h satisfies I, there exists a commit order co such that $\mathsf{wr} \cup \mathsf{so} \subseteq \mathsf{co}$ and $\langle h, \mathsf{co} \rangle$ satisfies the axioms defining I. We show that there exists an execution $c_0 c_1 \ldots c_n$ under $\Rightarrow_I$ where transactions are executed serially in the order defined by co, such that c_n is a final configuration that contains h. The only difficulty is showing that the read-extern transitions between two configurations c_j and c_{j+1} that add a write-read dependency $(t', \mathsf{read}(k, v)) \in \mathsf{wr}$ are enabled even though the transaction log t containing $\mathsf{read}(k, v)$ is "incomplete" in the history h_j of c_j, and h_j does not contain transactions committed after t. This relies on the prefix-closure property in Lemma 4.1. Let co_j be the order in which transactions have been executed until c_j. Then, $\langle h_j, \mathsf{co}_j \rangle$ is a prefix of $\langle h, \mathsf{co} \rangle$, and $\langle h_j, \mathsf{co}_j \rangle \models I$ because $\langle h, \mathsf{co} \rangle \models I$. □

It can also be shown that $\Rightarrow_I$ is *deadlock-free* for every natural isolation level (e.g., Read Committed, Causal Consistency, Snapshot Isolation, and Serializability), i.e., every read can return some value satisfying the axioms of I at the time when it is executed (independently of previous choices).

5 COMPILING SQL TO KEY-VALUE API

We define an operational semantics for SQL programs (in $\mathcal{P}_{SQL}$) based on a compiler that rewrites SQL queries to key-value `read` and `write` instructions. For presentation reasons, we use an intermediate representation where each table of a database instance is represented using a *set*-valued variable that stores values of the primary key (identifying uniquely the rows in the table) and a set of key-value pairs, one for each cell in the table.[7] In a second step, we define a rewriting of the API used to manipulate set variables into key-value `read` and `write` instructions.

Intermediate Representation. Let $\mathcal{S} : \mathbb{T} \rightharpoonup 2^{\mathbb{C}}$ be a database schema (recall that $\mathbb{T}$ and $\mathbb{C}$ are the set of table names and column names, resp.). For each table *tab*, let *tab.pkey* be the name of the primary key column. We represent an instance $\mathcal{D} : \mathsf{dom}(\mathcal{S}) \rightarrow 2^{\mathbb{R}}$ using:

- for each table *tab*, a set variable *tab* (with the same name) that contains the primary key value $r(tab.pkey)$ of every row $r \in \mathcal{D}(tab)$,
- for each row $r \in \mathcal{D}(tab)$ with primary key value $pkeyVal = r(tab.pkey)$, and each column $c \in \mathcal{S}(tab)$, a key *tab.pkeyVal.c* associated with the value $r(c)$.

Example 5.1. The table A on the left of Figure 8, where the primary key is defined by the Id column, is represented using a set variable A storing the set of values in the column Id, and one key-value pair for each cell in the table.

Figure 9 lists our rewriting of SQL queries over a database instance $\mathcal{D}$ to programs that manipulate the set variables and key-value pairs described above. This rewriting contains the minimal set of accesses to the cells of a table that are needed to implement an SQL query according to its conventional specification. To manipulate set variables, we use `add` and `remove` for adding and

[7] For simplicity, we assume that primary keys correspond to a single column in the table.

Table:

A		
Id	Name	City
1	Alice	Paris
2	Bob	Bangalore
3	Charles	Bucharest

Intermediate representation:

A = { 1, 2, 3 }

A.1.Id: 1, A.1.Name: Alice, A.1.City: Paris
A.2.Id: 2, A.2.Name: Bob, A.2.City: Bangalore
A.3.Id: 3, A.3.Name: Charles, A.3.City: Bucharest

Fig. 8. Representing tables with set variables and key-value pairs. We write a key-value pair as key:value.

```
SELECT/DELETE/UPDATE
   rows := elements(tab)
   for ( let pkeyVal of rows )
      for ( let c of c⃗2 )
         val[c] := read(tab.pkeyVal.c)
      if ( ϕ[c ↦ val[c] : c ∈ c⃗2] true )
         // SELECT c⃗1 AS x FROM tab WHERE ϕ(c⃗2)
         for ( let c of c⃗1 )
            out[c] := read(tab.pkeyVal.c)
         x := x ∪ out
         // DELETE FROM tab WHERE ϕ(c⃗2)
         remove(tab, pkeyVal);
         // UPDATE tab SET c⃗1 = x⃗ WHERE ϕ(c⃗2)
         for ( let c of c⃗1 )
            write( tab.pkeyVal.c, γ(x⃗[c]) )
```

```
INSERT INTO tab VALUES x⃗
   pkeyVal := γ(x⃗[0])
   if ( add(tab,pkeyVal) )
      for ( let c of S(tab) )
         write( tab.pkeyVal.c, γ(x⃗[c]) )
```

Fig. 9. Compiling SQL queries to the intermediate representation. Above, γ is a valuation of local variables. Also, in the case of INSERT, we assume that the first element of $\vec{x}$ represents the value of the primary key.

removing elements, respectively (returning true or false when the element is already present or deleted from the set, respectively), and `elements` that returns all of the elements in the input set.[8]

SELECT, DELETE, and UPDATE start by reading the contents of the set variable storing primary key values and then, for every row, the columns in $\vec{c_2}$ needed to check the Boolean condition ϕ (the keys corresponding to these columns). For every row satisfying this Boolean condition, SELECT continues by reading the keys associated to the columns that need to be returned, DELETE removes the primary key value associated to this row from the set *tab*, and UPDATE writes to the keys corresponding to the columns that need to be updated. In the case of UPDATE, we assume that the values of the variables in $\vec{x}$ are obtained from a valuation γ (this valuation would be maintained by the operational semantics of the underlying key-value store). INSERT adds a new primary key value to the set variable *tab* (the call to add checks whether this value is unique) and then writes to the keys representing columns of this new row.

Manipulating Set Variables. Based on the standard representation of a set using its characteristic function, we implement each set variable *tab* using a set of keys *tab*.has.*pkeyVal*, one for each value *pkeyVal* ∈ Vals. These keys are associated with Boolean values, indicating whether *pkeyVal* is contained in *tab*. In a concrete implementation, this set of keys need not be fixed a-priori, but can grow during the execution with every new instance of an INSERT. Figure 10 lists the implementations of add/`elements`, which are self-explanatory (`remove` is analogous).

[8] add(s, e) and remove(s, e) add and remove the element e from s, respectively. elements(s) returns the content of s.

```
add(tab,pkeyVal):
  if (read(tab.has.pkeyVal))
     return false
  write(tab.has.pkeyVal,true)
  return true
```

```
elements(tab):
  ret := ∅
  for ( let pkeyVal of Vals )
    if (read(tab.has.pkeyVal))
      ret := ret ∪ {pkeyVal}
  return ret;
```

Fig. 10. Manipulating set variables using key-value pairs.

6 IMPLEMENTATION

We implemented MonkeyDB to support an interface common to most storage systems. Operations can be either key-value (KV) updates (to access data as a KV map) or SQL queries (to access data as a relational database). MonkeyDB supports transactions as well; a transaction can include multiple operations. Figure 11 shows the architecture of MonkeyDB. A client can connect to MonkeyDB over a TCP connection, as is standard for SQL databases.[9] This offers a plug-and-play experience when using standard frameworks such as JDBC [Java Platform [n. d.]]. Client applications can also use MonkeyDB as a library in order to directly invoke the storage APIs, or interact with it via HTTP requests, with JSON payloads.

MonkeyDB contains a SQL-To-KV compiler that parses an input query, builds its Abstract Syntax Tree (AST) and then applies the rewriting steps described in Section 5 to produce an equivalent sequence of KV API calls (`read()` and `write()`).[10] It uses a hashing routine (`hash`) to generate unique keys corresponding to each cell in a table. For instance, in order to insert a value v for a column c in a particular row with primary key value *pkeyVal*, of a table *tab*, we invoke `write(hash(`*tab*`,` *pkeyVal*`,` c`),` v`)`. We currently support only a subset of the standard SQL operators. For instance, nested queries or join operators are unsupported; these can be added in the future with more engineering effort. Note that our SQL compiler maintains cell-level atomicity in order to minimize the set of possible conflicts between concurrent transactions (inspired by Rahmani et al. [2019]). If a particular SQL implementation requires row-level atomicity, it can be imposed in our compiler by allowing the keys to represent entire rows.

MonkeyDB schedules transactions from different sessions one after the other using a single global lock. Internally, it maintains execution state as a history consisting of a set of transaction logs, write-read relations and a partial session order (as discussed in §3). On a `read()`, MonkeyDB first collects a set of possible writes present in transaction log that can potentially form write-read (read-from) relationships, and then invokes the consistency checker (Figure 11) to confirm validity under the chosen isolation level. Finally, it randomly returns one of the values associated with valid writes. A user can optionally instruct MonkeyDB to only select from the set of *latest* valid write per session. This option helps limit weak behaviors for certain reads.

Our consistency checker maintains the session order and the write-read relation as a graph, and computes commit-order constraints based on the axioms described in Section 3.2. The consistency checker is an independent and pluggable module: we have one for Read Committed and one for Causal Consistency, and more can be added in the future. For these two isolation levels, we have developed an *incremental* version of the algorithms presented by Biswas and Enea [2019a] where commit-order constraints are preserved between different invocations of the consistency checker (as new operations/transactions are added to the history). This is possible because the commit-order constraints imposed by the corresponding axioms (Figure 3a and Figure 3b) depend only on

[9]We support the MySQL client-server protocol using https://github.com/jonhoo/msql-srv.
[10]We use https://github.com/ballista-compute/sqlparser-rs

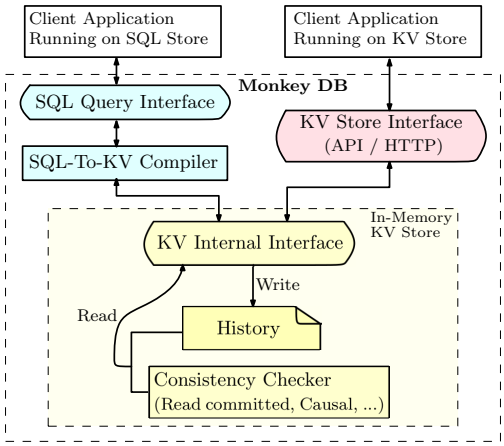


Fig. 11. Architecture of MonkeyDB

po, so, and wr, and these relations grow monotonically while the history is extended with new operations/transactions. While the consistency checker is prior work, it only constitutes 10% of the total code base (even excluding imported packages), signalling that significant implementation effort was required to design an efficient and usable tool.

7 EVALUATION: MICROBENCHMARKS

We consider a set of micro-benchmarks inspired from real-world applications (§7.1) and evaluate the number of test iterations required to fail an invalid assertion (§7.2). We also measure the *coverage* of weak behaviors provided by MonkeyDB (§7.3). Each of these applications were implemented based on their specifications described in prior work; they all use MonkeyDB as a library, via its KV interface.

7.1 Applications

Twitter [Twissandra 2020]. This is based on a social-networking application that allows users to create a new account, follow, unfollow, tweet, browse the newsfeed (tweets from users you follow) and the timeline of any particular user. Figure 12 shows the pseudo code for two operations, `NewsFeed` and `Timeline`.

A user can access twitter from multiple clients (sessions), which could lead to unexpected behavior under weak isolation levels. Consider the following scenario with two users, A and B where user A is accessing twitter from two different sessions, S_1 and S_2. User A views the timeline of user B from one session (S_1:`Timeline`(B)) and decides to follow B through another session (S_2:`Follow`(A, B)). Now when user A visits their timeline or newsfeed (S_2:`NewsFeed`(A)), they expect to see all the tweets of B that were visible via `Timeline` in session S_1. But under weak isolation levels, this does not always hold true and there could be missing tweets.

Shopping Cart [Sivaramakrishnan et al. 2015]. This application allows a user to add, remove and change quantity of items from different sessions. It also allows the user to view all items present in the shopping cart. The pseudo code and an unexpected behavior under weak isolation levels were discussed in §1, Figure 1.

Courseware [Nair et al. 2020]. This is an application for managing students and courses, allowing students to register, de-register and enroll for courses. Courses can also be created or deleted.

```
// Get following users' tweets
NewsFeed(user u) {
  begin()
  FW = read("following:" + u.id)
  NF = {}
  foreach v ∈ FW:
    T = read("tweets:" + v.id)
    NF = NF ∪ T
  commit()
  return sortByTime(NF)
}

// Get user's tweets
Timeline(user u) {
  begin()
  key = "tweets:" + u.id
  T = read(key)
  commit()
  return sortByTime(T)
}
```

```
Enroll(student s, course c) {
  begin()
  S = read("students")
  C = read("courses")
  if (s ∉ S || c ∉ C)
    throw InvalidEnrollment
  E = read("enrollments")
  if |{u: (u, c) ∈ E}| == capacity(c)
    throw CourseFull
  E = E ∪ {(s, c)}
  write("enrollments", E)
  commit()
}
```

```
Push(v) {
  n = {value: v, next: null}
  while (true)
    top = read("head");
    n.next = top;
    if (CAS("head", top, n))
      break
}

Pop() {
  while (true)
    top = read("head")
    if (top == null)
      return EMPTY
    v = top.value
    n = top.next
    if (CAS("head", top, n))
      return v
}
```

Fig. 12. Example operations from the microbenchmark applications

Courseware maintains the current status of students (registered, de-registered), courses (active, deleted) as well as enrollments. Enrollment can contain only registered students and active courses, subject to the capacity of the course. Figure 12 shows an implementation of the `Enroll` operation. `Enroll` checks that the given student and course are valid and also checks whether the course registration has reached its capacity before enrolling the student to the course.

Under weak isolation, it is possible that two different students, when trying to enroll concurrently, will both succeed even though only one spot was left in the course. Another example that breaks the application is when a student is trying to register for a course that is being concurrently removed: once the course is removed, no student should be seen as enrolled in that course.

Treiber Stack [Nagar et al. 2020]. Treiber stack is a concurrent stack data structure that uses compare-and-swap (CAS) instructions instead of locks for synchronization. This algorithm was ported to operate on a kv-store by Nagar et al. [2020] and we use that implementation. Essentially, the stack contents are placed in a kv-store, instead of using an in-memory linked data structure. The pseudo-code for push and pop operations is shown in Figure 12. Each row in the stack is represented by a key-value pair in the kv-store. The key is the node id and the value contains a pair consisting of the stack element and the key of the next row down in the stack. A designated key "head" stores the key of the top of the stack. We generate the node id based on the current size of the kv-store. Both push and pop operations use CAS to make changes to the "head" key-value pair. CAS is implemented as a transaction, but the pop and push operations do not use transactions, i.e., each read/write/CAS is its own transaction.

When two different clients try to pop from the stack concurrently, under serializability, each pop would return a unique value, assuming that each pushed value is unique. However, under Causal Consistency, concurrent pops can return the same value.

7.2 Assertion Checking

We ran the above applications with MonkeyDB to find out if assertions, capturing unexpected behavior, were violated under Causal Consistency. Table 1 summarizes the results. For each application, we used 3 client threads and 3 operations per thread. We ran each test, with MonkeyDB,

Table 1. Assertions checking results in microbenchmarks

Application	Assertion	Avg. time to fail (Iters)	(sec)
Stack	Element popped more than once	3.7	0.02
Courseware	Course registration overflow	10.6	0.09
Courseware	Removed course registration	57.5	0.52
Shopping	Item reappears after deletion	20.2	0.14
Twitter	Missing tweets in feed	6.3	0.03

10,000 times; we refer to a test run as an iteration. We report the average number of iterations (Iters) before an assertion failed, and the corresponding time taken (sec). All the assertions were violated within 58 iterations, in half a second or less. In contrast, running with an actual database almost never produces an assertion violation.

7.3 Coverage

The previous section only checked for a particular set of assertions. As an additional measure of test robustness, we count the number of distinct *client-observable states* generated by a test. A client-observable state, for an execution, is the vector of all values returned by read operations. For instance, a stack's state is defined by return values of `pop` operations; a shopping cart's state is defined by the return value of `GetCart` and so on.

For this experiment, we randomly generated test harnesses; each harness spawns multiple threads that each execute a sequence of operations. In order to compute the absolute maximum of possible states, we had to limit the size of the tests: either 2 or 3 threads, each choosing between 2 and 4 operations.

Note that any program that concurrently executes operations against a store has two main sources of non-determinism: the first is the interleaving of operations (i.e., the order in which operations are submitted to the store) and second is the choice of read-from (i.e., the value returned by the store under its configured isolation level). MonkeyDB only controls the latter; it is up to the application to control the former. There are many tools that systematically enumerate interleavings (such as Chess [Musuvathi and Qadeer 2008], Coyote [Microsoft Coyote 2019]), but we use a simple trick instead to avoid imposing any burden on the application: we included an option in MonkeyDB to deliberately add a small random delay (sleep between 0 and 4 ms) before each transaction begins. This option was sufficient in our experiments, as we show next.

We also implemented a special setup using the Coyote tool [Microsoft Coyote 2019] to enumerate all sources of non-determinism, interleavings as well as read-from, in order to explore the entire state space of a test. We use this to compute the total number of states. Figure 13 shows the number of distinct states observed under different isolation levels (with and without the delay option), averaged across multiple (50) test harnesses, as we increase the number of iterations. For demonstrating the effectiveness of MonkeyDB in the limit, we also show the max value computed by Coyote for each of serializability and Causal Consistency. Note that the Coyote approach was designed only to compute the max number. It usually took very large number of iterations to reach that max, hence we do not recommend it as a practically feasible approach.

Each of these graphs show similar trends: the number of states with Causal Consistency are *much higher* than with serializability. Thus, testing with a store that is unable to generate weak behaviors will likely be ineffective. Furthermore, the "delay" versions of MonkeyDB are able to approach the maximum within a few thousand attempts, implying that MonkeyDB's strategy of per-read randomness is effective for providing coverage to the application.

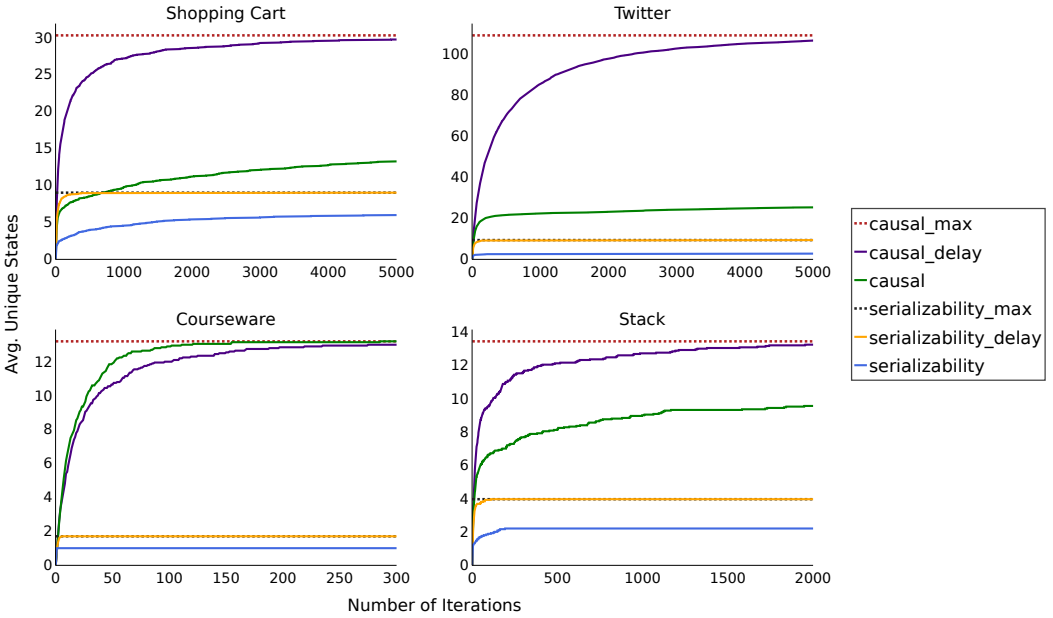


Fig. 13. State coverage obtained with MonkeyDB for various microbenchmarks

8 EVALUATION: OLTP WORKLOADS

OLTPBench [Difallah et al. 2013] is a benchmark suite of representative OLTP workloads for relational databases. We picked a subset of OLTPBench for which we had reasonable assertions. Table 2 lists basic information such as the number of database tables, the number of static transactions, how many of them are read-only, and the number of different assertions corresponding to system invariants for testing the benchmark. We modified OLTPBench by rewriting SQL join and aggregation operators into equivalent application-level loops, following a similar strategy as Rahmani et al. [2019]. Except for this change, we ran OLTPBench unmodified.

For TPC-C, we obtained a set of 12 invariants from its specification document [Council 2020]. For all other benchmarks, we manually identified invariants that the application should satisfy. We asserted these invariants by issuing a read-only transaction to MonkeyDB at the end of the execution of the benchmark. None of the assertions fail under serializability; they are indeed invariants under serializability.[11] When using weaker isolation, we configured MonkeyDB so that the reads in assertion-checking transactions return only latest valid writes per session (§6) in order to isolate the weak behavior to only the application.

We ran each benchmark 100 times and report, for each assertion, the number of runs in which it was violated. Note that OLTPBench runs in two phases. The first is a loading phase that consists of a big initial transaction to populate tables with data, and then the execution phase issues multiple concurrent transactions. With the goal of testing correctness, we *turn down* the scale factor to generate a small load and limit the execution phase time to ten seconds with just two or three sessions. A smaller test setup has the advantage of making debugging easier. With MonkeyDB, there is no need to generate large workloads.

[11]We initially observed two assertions failing under serializability. Upon analyzing the code, we identified that the behavior is due to a bug in OLTPBench that we have reported in an issue on their Github repository.

Table 2. OLTP benchmarks tested with MonkeyDB

Benchmark	#Tables	#Txns	#Read-only	#Assertions
TPC-C	9	5	2	12
SmallBank	3	6	1	1
Voter	3	1	0	1
Wikipedia	12	5	2	3

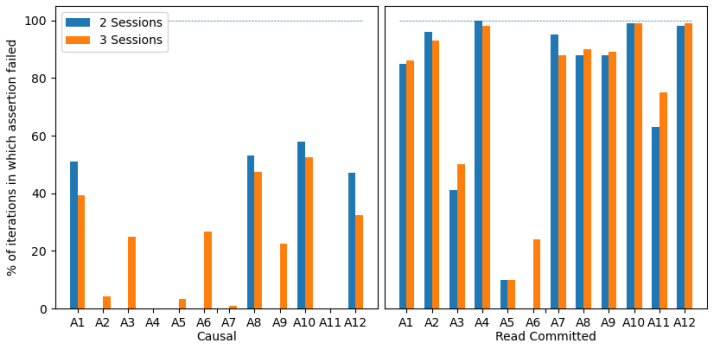


Fig. 14. Assertion checking with MonkeyDB: TPC-C

8.1 TPC-C

TPC-C emulates a wholesale supplier transactional system that delivers orders for a warehouse company. This benchmark deals with customers, payments, orders, warehouses, deliveries, etc. We configured OLTPBench to issue a higher proportion (> 85%) of update transactions, compared to read-only ones. Further, we considered a small input workload constituting of one warehouse, two districts per warehouse and three customers per district.

TPC-C has twelve assertions (A1 to A12) that check for consistency between the database tables. For example, A12 checks: for any customer, the sum of delivered order-line amounts must be equal to the sum of balance amount and YTD (Year-To-Date) payment amount of that customer.

Figure 14 shows the percentage of test runs in which an assertion failed. It shows that all the twelve assertions are violated under Read Committed isolation level. In fact, 9 out of the 12 assertions are violated in more than 60% of the test runs. Under Causal Consistency, all assertions are violated with three sessions, except for A4 and A11. We manually inspected TPC-C and we believe that both of these assertions are valid under Causal Consistency. For instance, A4 checks for consistency between two tables, both of which are only updated within the same transaction, thus Causal Consistency is enough to preserve consistency between them. These results demonstrate the effectiveness of MonkeyDB in breaking invalid assertions.

8.2 SmallBank, Voter, and Wikipedia

SmallBank is a standard financial banking system, dealing with customers, saving and checking accounts, money transfers, etc. Voter emulates the voting system of a television show and allows users to vote for their favorite contestants. Wikipedia is based on the popular online encyclopedia. It deals with a complex database schema involving page revisions, page views, user accounts, logging, etc. It allows users to edit its pages and maintains a history of page edits and user actions.

We identified a set of five assertions, A13 to A17, that should be satisfied by these systems. For SmallBank, we check if the total money in the bank remains the same while it is transfered from

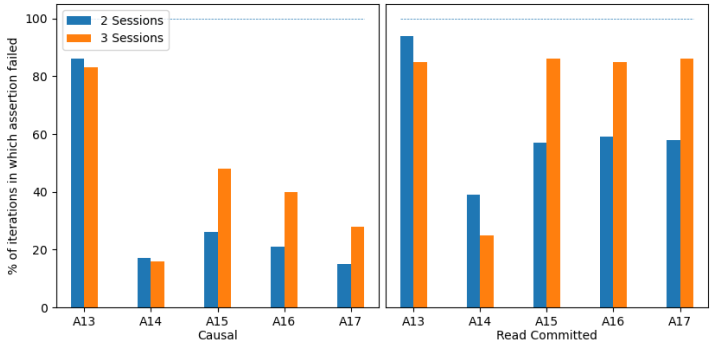


Fig. 15. Assertion checking: SmallBank, Voter, and Wikipedia

one account to another (A13). Voter requires that the number of votes by a user is limited to a fixed threshold (A14). For Wikipedia, we check if for a given user and for a given page, the number of edits recorded in the user information, history, and logging tables are consistent (A15-A17). As before, we consider small workloads: (1) five customers for SmallBank, (2) one user for Voter, and (3) two pages and two users for Wikipedia.

Figure 15 shows the results. MonkeyDB detected that all the assertions are invalid under the chosen isolation levels. Under Causal Consistency, MonkeyDB could break an assertion in 26.7% (geo-mean) runs given 2 sessions and in 37.2% (geo-mean) runs given 3 sessions. Under Read Committed, the corresponding numbers are 56.1% and 65.4% for 2 and 3 sessions, respectively.

9 COMPARISON TO OTHER TESTING TECHNIQUES

We compared the effectiveness of MonkeyDB against using a standard database (MySQL) and also against using the state-of-the-art random testing tool Jepsen [Jepsen 2020]. Jepsen injects noise, in the form of network and system faults, into an executing database, increasing the chances of exercising corner cases of the database. We conducted experiments using the TPC-C benchmark.

To run TPC-C on a standard database, we used the same test setup as in Section 8, except with MySQL in place of MonkeyDB. We configured MySQL to run under Read Committed and used the same execution time limit of 10 seconds, as that of MonkeyDB. We observed that using MySQL, only two assertions were violated (A10 and A12), even when we increased the number of sessions to ten. (Whereas, MonkeyDB could violate all twelve.) We note that MySQL is much faster than MonkeyDB. For 2 and 3 sessions under Read Committed with 10 seconds time limit, the total throughput (transactions per session) for MySQL is 450 and 300, respectively, whereas the throughput of MonkeyDB is 6 and 4, respectively. Nonetheless, MySQL is unable to violate most assertions.

For the Jepsen experiment, we configured a Galera Cluster [Galera 2020] to run MariaDB [Foundation 2020] with Jepsen. (We switched to using MariaDB instead of MySQL because it was easier to set up with Gelera, but MariaDB is designed to be a drop-in replacement for MySQL, so we do not expect results to change much between these two databases.) We configured MariaDB to use Read Committed. With Jepsen, we cannot run OLTPBench directly because Jepsen requires the workload to be expressed in its own language (Clojure). We used a version of TPC-C transactions written in Java (that can be easily invoked from Clojure) from prior work [Rahmani 2018; Rahmani et al. 2019]. We still had to initialize the database. For this, we created a dump of the database state

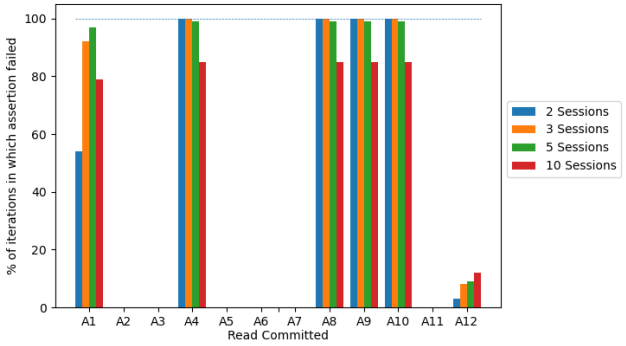


Fig. 16. Assertion checking with Jepsen (Bridge nemesis): TPC-C

using `sqldump`[12] after the initialization phase of OLTPBench finished. We loaded this dump into MariaDB before running the TPC-C transactions.

We conducted the experiment on different cluster sizes: two, three, five and ten nodes, with one session per node. We let Jepsen run for the same execution time limit of 10 seconds with 20 transactions per second on average. Jepsen provides different fault injection strategies (called *nemesis*). We ran our experiment with three different nemesis configurations [Jepsen Nemesis 2021]: (1) 'clock scrambler' with two seconds (scrambles the system clock at each node), (2) 'partition-random-halves' (random partitioning of the network into two halves), and (3) 'bridge' (random partitioning as in the previous configuration but with an additional bridging node). We noticed that the Galera cluster is robust against these different nemesis configurations, and the number of assertion violations remains the same for each of them. Figure 16 shows the percentage of test runs in which the assertions are violated when TPC-C is run for 100 times on Jepsen with 'bridge' configuration. It shows that the Jepsen setup could violate only six out of twelve assertions, even when run with up to ten sessions.

The above results demonstrate the ability of MonkeyDB to efficiently test for weak behaviours in client applications in comparison with related techniques. In contrast to Jepsen, MonkeyDB does not require setting up a cluster, trying out different partitioning heuristics and system configurations to introduce weak behaviours, or re-writing client applications. MonkeyDB makes it straightforward to unit test any storage-backed application.

We would like to qualify the above results with a remark. MonkeyDB works off the specification of an isolation level, rather than a concrete implementation. It is possible that some implementations (like MySQL or MariaDB) may not exhibit all possible weak behaviors for a declared isolation level, i.e., they implement something stronger. In other words, MonkeyDB is a realization of the specification rather than a replacement for a concrete implementation. Therefore, it is possible, that in the experiments described above, it may, in fact, be impossible for the remaining TPC-C assertions to be violated with MariaDB because MariaDB implements something stronger than Read Committed.

However, our results are still useful. Knowing the exact isolation guarantees of a database requires a close examination of its implementation that a user may not be prepared to do. Moreover, implementations change with software versions, and some applications need to support multiple databases in a plug-and-play manner. For the developer, it is thus still useful to work off the

[12] Available here: https://mariadb.com/kb/en/mysqldump/

specification of the isolation level, rather than be left wondering about actual behaviors that a database might provide.

In terms of the formalizations, we have followed Berenson et al. [1995] and Cerone et al. [2015]. In particular, Biswas and Enea [2019b] has shown that the axiomatic models we are using in this paper are equivalent to those of Cerone et al. [2015]. It is an important problem for the research community to define *tight* formalizations for existing databases, but that problem is beyond the scope of this paper.

10 RELATED WORK

There have been several directions of work addressing the correctness of database-backed applications. We directly build upon one line of work concerned with the logical formalization of isolation levels [Adya et al. 2000; Berenson et al. 1995; Biswas and Enea 2019a; Cerone et al. 2015; X3 1992]. Our work relies on the axiomatic definitions of isolation levels introduced by Biswas and Enea [2019a], which have also investigated the problem of checking whether a given history satisfies a certain isolation level. Our kv-store implementation relies on these algorithms to check the validity of the values returned by read operations. Working with a logical formalization allowed us to avoid implementing an actual database with replication or sophisticated synchronization.

Another line of work concentrates on the problem of finding "anomalies": behaviors that are not possible under serializability. This is typically done via a static analysis of the application code that builds a static dependency graph that over-approximates the data dependencies in all possible executions of the application [Bernardi and Gotsman 2016; Cerone and Gotsman 2018; Fekete et al. 2005; Gan et al. 2020; Jorwekar et al. 2007; Warszawski and Bailis 2017]. Anomalies with respect to a given isolation level then correspond to a particular class of cycles in this graph. Static dependency graphs turn out to be highly imprecise in representing feasible executions, leading to false positives. Another source of false positives is that an anomaly might not be a bug because the application may already be designed to handle the non-serializable behavior [Brutschy et al. 2018; Gan et al. 2020]. Recent work has tried to address these issues by using more precise logical encodings of the application, e.g. [Brutschy et al. 2017, 2018], or by using user-guided heuristics [Gan et al. 2020].

Another approach consists of modeling the application logic and the isolation level in first-order logic and relying on SMT solvers to search for anomalies [Kaki et al. 2018; Nagar and Jagannathan 2018; Ozkan 2020], or defining specialized reductions to assertion checking [Beillahi et al. 2019a,b]. The Clotho tool [Rahmani et al. 2019], for instance, uses a static analysis of the application to generate test cases with plausible anomalies, which are deployed in a concrete testing environment for generating actual executions. Our approach, based on testing with MonkeyDB, has several practical advantages. There is no need for analyzing application code; we can work with any application. There are no false positives because we directly run the application and check for user-defined assertions, instead of looking for application-agnostic anomalies. The limitation, however, of the MonkeyDB approach is the inherent incompleteness of testing.

Several works have looked at the problem of reasoning about the correctness of applications executing under weak isolation and introducing additional synchronization when necessary [Balegas et al. 2015; Gotsman et al. 2016; Li et al. 2014; Nair et al. 2020]. As in the previous case, our work based on testing has the advantage that it can scale to real sized applications (as opposed to these techniques which are based on static analysis or logical proof arguments), but it cannot prove that an application is correct. Moreover, the issue of repairing applications is orthogonal to our work.

From a technical perspective, our operational semantics based on recording past operations and certain data-flow and control-flow dependencies is similar to recent work on stateless model checking in the context of weak memory models (of hardware or languages). One line of work, e.g. [Abdulla et al. 2015; Kokologiannakis et al. 2018], is concerned with partial-order reduction

(POR) so that the model checkers can avoid enumerating equivalent executions. MonkeyDB instead relies on randomness to provide coverage. POR is beyond the scope of our work, but an interesting direction for future work. Another similarity is with testing of C/C++ applications under the c11 memory model [Lidbury and Donaldson 2017; Norris and Demsky 2013]. DB applications are, however, very different in nature compared to ones that exploit the c11 memory model. The latter is about implementation of mutual exclusion, concurrent lock-free data structures, etc., which are usually not a concern with DB applications that instead rely on transactions. Furthermore, a DB is usually always external to an application, allowing us to design MonkeyDB as a standalone entity. With C/C++, each load or store operation potentially interacts with the memory model, mandating the need for detailed instrumentation techniques.

11 CONCLUSIONS AND FUTURE WORK

Our goal is to enable developers to test the correctness of their storage-backed applications under weak isolation levels. Such bugs are hard to catch because weak behaviors are rarely generated by real storage systems, but failure to address them can lead to loss of business [Warszawski and Bailis 2017]. We present MonkeyDB, an easy-to-use mock storage system for weeding out such bugs. MonkeyDB uses a logical understanding of isolation levels to provide (randomized) coverage of all possible weak behaviors. Our evaluation reveals that using MonkeyDB is very effective at breaking assertions that would otherwise hold under a strong isolation level.

In future work, we would like to explore strategies other than just random selection to resolve read operations, for instance, biasing towards "older" values, or enumerative techniques that guarantee a distinct behavior on each run. So far though, random sets a strong baseline. Another interesting line of work would be to support workloads where transactions can have *different* isolation levels. This first requires a formalization of the semantics of mixed isolation levels that is faithful to existing implementations. An axiomatic formalization should then be easy to integrate with MonkeyDB.

ACKNOWLEDGMENTS

This work is supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 678177).

REFERENCES

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9035)*, Christel Baier and Cesare Tinelli (Eds.). Springer, 353–367. https://doi.org/10.1007/978-3-662-46681-0_28

A. Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Technical Report. USA.

Atul Adya, Barbara Liskov, and Patrick E. O'Neil. 2000. Generalized Isolation Level Definitions. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, David B. Lomet and Gerhard Weikum (Eds.). IEEE Computer Society, 67–78. https://doi.org/10.1109/ICDE.2000.839388

Deepthi Devaki Akkoorath and Annette Bieniusa. 2016. *Antidote: the highly-available geo-replicated database with strongest guarantees.* Technical Report. https://pages.lip6.fr/syncfree/attachments/article/59/antidote-white-paper.pdf

Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno M. Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, Laurent Réveillère, Tim Harris, and Maurice Herlihy (Eds.). ACM, 6:1–6:16. https://doi.org/10.1145/2741948.2741972

Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2019a. Checking Robustness Against Snapshot Isolation. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019,*

Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11562), Isil Dillig and Serdar Tasiran (Eds.). Springer, 286–304. https://doi.org/10.1007/978-3-030-25543-5_17

Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. 2019b. Robustness Against Transactional Causal Consistency. In *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands (LIPIcs, Vol. 140)*, Wan J. Fokkink and Rob van Glabbeek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:18. https://doi.org/10.4230/LIPIcs.CONCUR.2019.30

Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 1–10. https://doi.org/10.1145/223784.223785

Giovanni Bernardi and Alexey Gotsman. 2016. Robustness against Consistency Models with Atomic Visibility. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada (LIPIcs, Vol. 59)*, Josée Desharnais and Radha Jagadeesan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:15. https://doi.org/10.4230/LIPIcs.CONCUR.2016.7

Ranadeep Biswas and Constantin Enea. 2019a. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 165:1–165:28. https://doi.org/10.1145/3360591

Ranadeep Biswas and Constantin Enea. 2019b. On the Complexity of Checking Transactional Consistency. *CoRR* abs/1908.04509 (2019). arXiv:1908.04509 http://arxiv.org/abs/1908.04509

Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. 2021a. MonkeyDB: Effectively Testing Correctness against Weak Isolation Levels. *CoRR* abs/2103.02830 (2021). arXiv:2103.02830 https://arxiv.org/abs/2103.02830

Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. 2021b. *MonkeyDB: Effectively Testing Correctness under Weak Isolation Levels.* https://doi.org/10.5281/zenodo.5504052

Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, Andrew Birrell and Emin Gün Sirer (Eds.). USENIX Association, 49–60. https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

Lucas Brutschy, Dimitar I. Dimitrov, Peter Müller, and Martin T. Vechev. 2017. Serializability for eventual consistency: criterion, analysis, and applications. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 458–472. http://dl.acm.org/citation.cfm?id=3009895

Lucas Brutschy, Dimitar I. Dimitrov, Peter Müller, and Martin T. Vechev. 2018. Static serializability analysis for causal consistency. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 90–104. https://doi.org/10.1145/3192366.3192415

Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015 (LIPIcs, Vol. 42)*, Luca Aceto and David de Frutos-Escrig (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 58–71. https://doi.org/10.4230/LIPIcs.CONCUR.2015.58

Andrea Cerone and Alexey Gotsman. 2018. Analysing Snapshot Isolation. *J. ACM* 65, 2 (2018), 11:1–11:41. https://doi.org/10.1145/3152396

Transaction Processing Performance Council. 2020. TPC-C Benchmark Specification. http://tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, Thomas C. Bressoud and M. Frans Kaashoek (Eds.). ACM, 205–220. https://doi.org/10.1145/1294261.1294281

Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (2013), 277–288. https://doi.org/10.14778/2732240.2732246

Alan D. Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis E. Shasha. 2005. Making snapshot isolation serializable. *ACM Trans. Database Syst.* 30, 2 (2005), 492–528. https://doi.org/10.1145/1071610.1071615

MariaDB Foundation. 2020. MariaDB: An Open Source Relational Database. https://mariadb.org/.

Galera. 2020. Galera Cluster for MySQL. https://galeracluster.com/.

Yifan Gan, Xueyuan Ren, Drew Ripberger, Spyros Blanas, and Yang Wang. 2020. IsoDiff: Debugging Anomalies Caused by Weak Isolation. *Proc. VLDB Endow.* 13, 12 (July 2020), 27732786. https://doi.org/10.14778/3407790.3407860

Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm strong enough: reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 371–384. https://doi.org/10.1145/2837614.2837625

Java Platform. [n. d.]. JDBC: Java Database Connectivity API. https://en.wikipedia.org/wiki/Java_Database_Connectivity.

Jepsen. 2020. Distributed Systems Testing. https://jepsen.io/.

Jepsen Nemesis. 2021. Tutorial on Jepsen nemesis variants. https://jepsen-io.github.io/jepsen/jepsen.nemesis.html.

Sudhir Jorwekar, Alan D. Fekete, Krithi Ramamritham, and S. Sudarshan. 2007. Automating the Detection of Snapshot Isolation Anomalies. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold (Eds.). ACM, 1263–1274. http://www.vldb.org/conf/2007/papers/industrial/p1263-jorwekar.pdf

Gowtham Kaki, Kapil Earanky, K. C. Sivaramakrishnan, and Suresh Jagannathan. 2018. Safe replication through bounded concurrency verification. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 164:1–164:27. https://doi.org/10.1145/3276534

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2018. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.* 2, POPL (2018), 17:1–17:32. https://doi.org/10.1145/3158105

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

Cheng Li, João Leitão, Allen Clement, Nuno M. Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, 281–292. https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2

Christopher Lidbury and Alastair F. Donaldson. 2017. Dynamic race detection for C++11. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 443–457. https://doi.org/10.1145/3009837.3009857

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, Ted Wobber and Peter Druschel (Eds.). ACM, 401–416. https://doi.org/10.1145/2043556.2043593

Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 295310. https://doi.org/10.1145/2815400.2815426

Microsoft. 2020. Azure Cosmos DB Local Emulator. https://docs.microsoft.com/en-us/azure/cosmos-db/local-emulator.

Microsoft Coyote. 2019. Fearless coding for reliable asynchronous software. https://github.com/microsoft/coyote.

Madanlal Musuvathi and Shaz Qadeer. 2008. Fair stateless model checking. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 362–371. https://doi.org/10.1145/1375581.1375625

Kartik Nagar and Suresh Jagannathan. 2018. Automated Detection of Serializability Violations Under Weak Consistency. In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China (LIPIcs, Vol. 118)*, Sven Schewe and Lijun Zhang (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 41:1–41:18. https://doi.org/10.4230/LIPIcs.CONCUR.2018.41

Kartik Nagar, Prasita Mukherjee, and Suresh Jagannathan. 2020. Semantics, Specification, and Bounded Verification of Concurrent Libraries in Replicated Systems. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12224)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 251–274. https://doi.org/10.1007/978-3-030-53288-8_13

Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. 2020. Proving the Safety of Highly-Available Distributed Objects. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 544–571. https://doi.org/10.1007/978-3-030-44914-8_20

Brian Norris and Brian Demsky. 2013. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 131–150. https://doi.org/10.1145/2509136.2509514

Burcu Kulahcioglu Ozkan. 2020. Verifying Weakly Consistent Transactional Programs Using Symbolic Execution. In *Networked Systems - 8th International Conference, NETYS 2020, Marrakech, Morocco, June 3-5, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12129)*, Chryssis Georgiou and Rupak Majumdar (Eds.). Springer, 261–278. https:

//doi.org/10.1007/978-3-030-67087-0_17

Christos H. Papadimitriou. 1979. The serializability of concurrent database updates. *J. ACM* 26, 4 (1979), 631–653. https://doi.org/10.1145/322154.322158

Andrew Pavlo. 2017. What Are We Doing With Our Lives? Nobody Cares About Our Concurrency Control Research. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 3. https://doi.org/10.1145/3035918.3056096

Jos Rolando Guay Paz. 2018. *Microsoft Azure Cosmos DB Revealed: A Multi-Modal Database Designed for the Cloud* (1st ed.). Apress, USA.

Kia Rahmani. 2018. Lifting Jepsen Tests to the Application Level. https://kiarahmani.github.io/posts/2018/12/jepsen/.

Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. 2019. CLOTHO: Directed Test Generation for Weakly Consistent Database Systems. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 117 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360543

K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 413–424. https://doi.org/10.1145/2737924.2737981

Twissandra. Accessed November 1, 2020. *Twitter clone on Cassandra.* https://github.com/twissandra/twissandra

Todd Warszawski and Peter Bailis. 2017. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 520. https://doi.org/10.1145/3035918.3064037

ANSI X3. 1992. *135-1992. American National Standard for Information Systems-Database Language-SQL.* Technical Report.